Getting started on an HPC cluster

Ahmad Ammar

16 November 2026

Contents

1 What a cluster is **4**

1.1 The idea . . . 4

1.2 Do you need one? . . . 4

1.3 Kelvin2 in brief . . . 5

2 The command line **6**

2.1 Anatomy of a command . . . 6

2.2 Paths . . . 6

2.3 Commands worth knowing . . . 6

2.4 Joining commands . . . 7

2.5 Variables and loops . . . 7

2.6 Windows line endings . . . 8

3 Connecting securely **9**

3.1 How SSH keys work . . . 9

3.2 Step 1: Make a key pair . . . 9

3.3 Step 2: Give the cluster your public key . . . 9

3.4 Step 3: Log in, and set up MFA . . . 9

3.5 Step 4: Make a shortcut . . . 10

3.6 Step 5: Optional: stop retyping the passphrase . . . 10

4 Files and storage **11**

4.1 Where things go . . . 11

4.2 A tidy project . . . 11

4.3 Moving files with scp . . . 11

4.4 Many small files: pack first . . . 12

4.5 Large transfers . . . 12

4.6 Files that arrive read-only . . . 12

5 Software, environments and containers **13**

5.1 Step 1: Modules . . . 13

5.2 Step 2: Conda environments . . . 13

5.3 Step 3: Compiling from source . . . 14
5.4 Step 4: Containers . . . 15
5.4.1 What containerisation is . . . 15
5.4.2 Why not Docker on a cluster? . . . 15

6 Running jobs with Slurm **16**
6.1 Two ways to run . . . 16
6.2 A first job . . . 16
6.3 The settings you need . . . 16
6.4 Watching and managing . . . 17
6.5 Start small . . . 17

7 Seven exercises to build confidence **18**
7.1 Get the files . . . 18
7.2 Exercise 1: Hello from a compute node . . . 18
7.3 Exercise 2: Run a small program . . . 19
7.4 Exercise 3: Ask for more cores . . . 19
7.5 Exercise 4: Analyse real data . . . 20
7.6 Exercise 5: Your first job array . . . 20
7.7 Exercise 6: A geospatial calculation . . . 21
7.8 Exercise 7: Satellite tiles in parallel . . . 22
7.9 From exercises to your own project . . . 22

8 Job arrays: a parameter sweep **23**
8.1 The science in one paragraph . . . 23
8.2 Step 1: the program . . . 23
8.3 Step 2: make the inputs . . . 24
8.4 Step 3: the job script . . . 25
8.5 Step 4: how many tasks? . . . 25
8.6 Step 5: did it all work? . . . 26
8.7 Step 6: collect the results . . . 26
8.8 Choosing a packing . . . 26

9 Measuring and good practice **28**
9.1 Words for time . . . 28
9.2 Slurm's accounts . . . 28

9.3 A measured example . . . 28
9.4 Benchmarking fairly . . . 29
9.5 Good habits . . . 29
9.6 Being a good neighbour . . . 29

10 Other clusters, the cloud and Linux basics **31**
10.1 A checklist for any new system . . . 31
10.2 Other schedulers . . . 31
10.3 Cluster or cloud? . . . 31
10.4 Linux basics for looking after your own work . . . 32
10.4.1 Permissions . . . 32
10.4.2 Space . . . 32
10.4.3 Processes . . . 32
10.4.4 Sessions that survive a disconnection . . . 33
10.4.5 Scheduled tasks . . . 33
10.4.6 Installing system software . . . 33
10.4.7 Keeping your own server safe . . . 33

11 Troubleshooting and cheat sheet **34**
11.1 Errors and what they mean . . . 34
11.2 Cheat sheet . . . 34
11.3 References . . . 35

1 What a cluster is

1.1 The idea

Picture a laptop as one cook in one kitchen. A cluster is a building with hundreds of kitchens, each with dozens of cooks. It rarely makes one cook much faster. What it gives you is the ability to cook thousands of dishes at once, or to share out one enormous dish. There is a reception desk where you hand in orders, and a head waiter who decides which kitchen is free.

Word	Meaning	In the picture
Node	One computer in the cluster	A kitchen
Core	One independent processing unit in a node; it runs one stream of instructions at a time	A cook
Memory (RAM)	Fast working space that empties when the job ends	The worktop
Login node	The computer you arrive at. For editing, moving files and submitting work	Reception
Compute node	Where jobs run	The kitchens
Scheduler	Software that queues jobs and allocates resources. Slurm (Yoo et al., 2003) is widely used	The head waiter
Partition	A named queue of nodes with its own rules	A type of kitchen
Job	One piece of work submitted to the scheduler	An order

The login node is shared with everyone. **Do not run heavy work there**: it slows the system for others, and your process may be killed.

1.2 Do you need one?

A cluster helps when your work is too big, too long or too repetitive for your computer. It will not speed up one ordinary program that can only use a single core, beyond any difference in the processor. Three common patterns:

Pattern	Example	What it needs
Many independent runs, called "many-task" (Raicu et al., 2008) or "embarrassingly parallel"	A parameter sweep, or one analysis per file	Job arrays. Often no code changes
One program sharing memory on a node	A threaded model, or machine learning on many cores	`--cpus-per-task`, and a program that can use them
One program across nodes	A large fluid simulation, using MPI	`--ntasks`, `--nodes`, and an MPI-aware program

Asking for more cores does not by itself make a program use them. Even a parallel program has limits: the part that cannot be shared sets a ceiling on the speed-up, an idea known as Amdahl's law (Amdahl,

1967). If 10% of the work must run in series, then no number of cores can make it more than 10 times faster.

This guide concentrates on the first pattern, which is what many researchers need first. The habits in Wilson et al. (2014) are worth reading too.

1.3 Kelvin2 in brief

Kelvin2 only. The official pages[1] describe about 100 standard compute nodes with 128 cores each (two AMD EPYC 7702 processors) and nearly 800 GB of memory, 8 high-memory nodes, GPU nodes, and 2 PB of scratch storage, joined by InfiniBand networking. Figures differ slightly between NI-HPC's pages and change over time. To see what exists now, use `sinfo`.

Access is for researchers at Queen's University Belfast and Ulster University, and others through national schemes: see NI-HPC's access page[2]. The acceptable use policy[3] is on the documentation home page. Read it first.

[1] `https://www.qub.ac.uk/sites/ni-hpc/Kelvin2/`
[2] `https://www.qub.ac.uk/sites/ni-hpc/Kelvin2/Access/`
[3] `https://ni-hpc.github.io/nihpc-documentation`

2 The command line

A **shell** reads what you type and runs it. On Windows you will probably use PowerShell to connect; on the cluster, you will be using `bash`. macOS and Linux terminals work almost the same as `bash`. For a longer course, see Shotts (2019) or Software Carpentry[1].

2.1 Anatomy of a command

```
# cluster
ls -l ~/project
```

`ls` is the command: what to do (list files). `-l` is an option: how to do it (long format). `~/project` is the argument: what to act on. Spaces separate the parts, so a path containing a space needs double quotes, or the shell reads it as two things.

2.2 Paths

Symbol	Meaning
`~`	Your home folder
`.` and `..`	This folder, and the one above
`/`	Top of the Linux file tree, and the separator between folders
`*`	Wildcard: `case_*.json` matches every such file
`$HOME`, `$USER`	Variables that hold your home folder and username

Linux names are case-sensitive, so `Data` and `data` differ. Windows writes `C:\Users\me`, and Linux writes `/users/me`.

2.3 Commands worth knowing

Command	What it does
`pwd`	Print the folder you are in
`ls`, `ls -l`, `ls -t`	List files; with details; newest first
`cd folder`, `cd ..`, `cd`	Go to a folder; up one; home
`mkdir -p a/b`	Make folders, including parents, with no error if they exist
`cp a b`, `cp -r a b`	Copy a file; `-r` copies a folder and its contents
`mv a b`	Move or rename
`rm file`	Delete. **There is no recycle bin.** Look first with `ls`
`ln -s target name`	Make a shortcut (a symbolic link)
`cat`, `head -5`, `tail -20`	Print a whole file; its first 5 lines; its last 20
`less file`	Read a long file page by page; `q` quits
`wc -l`	Count lines
`grep "text" file`	Show lines containing text. `-i` ignores case, `-l` lists only file names

[1] `https://swcarpentry.github.io/shell-novice/`

Command	What it does
`sed -i 's/old/new/' file`	Replace text in a file
`chmod u+x file`	Let yourself (`u`) execute (`x`) a file. `chmod -R u+w folder` lets you write to everything in a folder
`tar czf a.tgz folder`	Pack a folder into one compressed archive (create, gzip, file)
`tar xzf a.tgz`	Unpack it (extract)
`df -h .`, `du -sh folder`	Free space on this disk; size of a folder
`hostname`, `nproc`	Which computer am I on? How many cores may I use?
`man ls`	The manual for a command; `q` quits

To edit on the cluster, `nano file` is the easiest editor if it is installed: the keys are listed at the bottom. NI-HPC's guide uses `vi`, which is powerful but odd at first.

2.4 Joining commands

```
# cluster

# | (a pipe) feeds one command's output into the next: count the log files
ls logs/*.out | wc -l

# > saves output to a file, replacing what was there; >> adds to the end
ls -l > listing.txt

# && runs the second command only if the first succeeded
mkdir -p results && cd results

# ; runs the second command whatever happened to the first
cd results; ls

# 2>/dev/null throws away error messages (stream 2) instead of printing them
ls /no/such/folder 2>/dev/null

# $( ) puts one command's output inside another: show the newest log
cat "$(ls -t logs/*.out | head -1)"
```

2.5 Variables and loops

```
# cluster

# Set a variable. No spaces around the =
DATA=/mnt/scratch2/users/$USER/data

# Use it with $. Double quotes keep it in one piece even if it contains a space
ls "$DATA"

# A for loop: repeat for each folder; $d holds the current one
for d in results/*/; do
  # basename strips the path; wc -l counts the files listed
  echo "$(basename "$d"): $(ls "$d" | wc -l) files"
done
```

2.6 Windows line endings

Windows ends each line of text with two invisible characters (carriage return, line feed), and Linux with one. A script written on Windows can fail on Linux with baffling errors such as `$'\r': command not found`. Fix it on the cluster:

```
# cluster
# In each line, replace a carriage return (\r) at the end of the line ($) with nothing
sed -i 's/\r$//' myscript.sh
```

3 Connecting securely

3.1 How SSH keys work

SSH (Secure Shell) is an encrypted remote terminal (OpenSSH, n.d.). Instead of a password, it can use a matched pair of files. The **public key** is like a padlock: you can give out copies, and the cluster fits one to your account. The **private key** is the only thing that opens it, and it stays on your computer. A **passphrase** locks the private key in case your laptop is stolen.

Never share your private key, its passphrase or your MFA secret with anybody, including IT staff or a colleague. Only the file ending in `.pub` is safe to send.

3.2 Step 1: Make a key pair

```
# laptop (PowerShell on Windows 10 or later; on macOS or Linux, use ~/.ssh/ instead)

# Make the folder that SSH looks in. -Force means no error if it exists
mkdir $HOME\.ssh -Force

# -t chooses the key type, -b its size in bits, -f the file name
ssh-keygen -t rsa -b 4096 -f $HOME\.ssh\my-kelvin-key
```

Set a passphrase when asked: Kelvin2 requires one. Nothing appears as you type, which is normal. This creates `my-kelvin-key` (private) and `my-kelvin-key.pub` (public). A forgotten passphrase cannot be recovered: you have to make a new pair.

3.3 Step 2: Give the cluster your public key

Each cluster has its own process. **Kelvin2 only:** apply through the access page[1]; the documentation says to allow 48 hours. If you are outside the QUB network, as Ulster University users are, you send NI-HPC the public key, as set out in their connection guide[2].

```
# laptop
# Copy the public key to the clipboard, ready to paste
Get-Content $HOME\.ssh\my-kelvin-key.pub | Set-Clipboard
```

3.4 Step 3: Log in, and set up MFA

```
# laptop
# -p: network port. -i: which private key. Then username@address
ssh -p 55890 -i $HOME\.ssh\my-kelvin-key <username>@login.kelvin.alces.network
```

Kelvin2 only. Port 55890 and this address are for connecting from outside the QUB network. From inside it, the address is `kelvin2.qub.ac.uk` on the normal port. MFA is mandatory. On first login, generate it:

[1] `https://www.qub.ac.uk/sites/ni-hpc/Kelvin2/Access/`
[2] `https://ni-hpc.github.io/nihpc-documentation/Connecting%20to%20Kelvin2/`

```
# cluster
/opt/flight/bin/flight mfa generate
```

Scan the QR code into an authenticator app. From then on, each login asks for `Verification code:`. If you lose the app, NI-HPC can reset it.

You land on one of four login nodes, shown in your prompt: `[<username>@login1 [kelvin2] ~]$`. The first time you reach a new login node you may see a warning about the host's identity. The connection guide explains what to do; read it rather than clicking through.

3.5 Step 4: Make a shortcut

The file `.ssh/config` on your laptop stores settings under a nickname. Open it with `notepad $HOME\.ssh\config`, and add the lines below. Comments must be on their own line in this file.

```
# laptop: .ssh/config
# The nickname you will type
Host kelvin
  # The real address
  HostName login.kelvin.alces.network
  Port 55890
  User <username>
  # Your private key
  IdentityFile ~/.ssh/my-kelvin-key
  # Send a small signal every 60 seconds, so that idle sessions stay up
  ServerAliveInterval 60
```

Now `ssh kelvin` is enough, and `scp` understands `kelvin:` as well.

3.6 Step 5: Optional: stop retyping the passphrase

```
# laptop (the first line may need an administrator PowerShell)

# Have the Windows SSH agent start automatically
Get-Service ssh-agent | Set-Service -StartupType Automatic
Start-Service ssh-agent

# Hand it your key: it holds the unlocked key in memory
ssh-add $HOME\.ssh\my-kelvin-key
```

You will still need the MFA code. `exit` ends a session.

If you prefer a graphical program on Windows, NI-HPC's guide also covers MobaXterm and PuTTY.

4 Files and storage

4.1 Where things go

Clusters have several storage areas with different rules. Learn yours before you create anything big.

Kelvin2 only, according to the quick-start guide[1]:

Area	Path	Limit	Deletion
Home	`/users/$USER`	50 GB and 100k files	None automatic
Scratch	`/mnt/scratch2/users/$USER`	No per-user quota	**After 90 days without access**
Temp	`/tmp` on each node	The node's disk	When the session ends
McClayRDS	Project paths	By project	QUB projects only

`quota -s` shows your actual allocation.

The key rule: **scratch is not a backup and not an archive.** Keep a copy of results that matter on your institution's storage, and ask your support team what, if anything, is backed up.

4.2 A tidy project

```
# cluster

# Braces expand: this makes all four folders
mkdir -p ~/project/{code,jobs,logs,envs}

# Big data lives on scratch
mkdir -p /mnt/scratch2/users/$USER/project/{data,outputs,tmp}

# A shortcut, so ~/project/scratch stands for the long path
ln -s /mnt/scratch2/users/$USER/project ~/project/scratch
```

4.3 Moving files with scp

`scp source destination` copies securely. A remote location is written `kelvin:path`, using your shortcut. **Run it from your laptop**, in a separate terminal from your cluster session.

```
# laptop

# Upload a file. Quotes are needed because of the space in the path
scp "D:\My Work\data.csv" kelvin:~/project/scratch/data/

# Upload a folder: -r means recursive
scp -r "D:\My Work\code" kelvin:~/project/

# Download: the source is now the remote one
scp kelvin:~/project/scratch/outputs/results.tgz "D:\My Work"
```

[1] `https://ni-hpc.github.io/nihpc-documentation/Quickstart%20Guide/`

Without the shortcut, the port needs a **capital** `-P`, unlike `ssh`, which uses a lower-case `-p`:

```
# laptop
scp -P 55890 -i $HOME\.ssh\my-kelvin-key data.csv <username>@login.kelvin.alces.network:~/project/
```

A Windows trap. Do not end a quoted folder with a backslash. In `"D:\My Work\"`, the `\"` is read as a literal quote mark, and you get a confusing "No such file or directory". Write `"D:\My Work"`.

4.4 Many small files: pack first

Sending thousands of small files one by one is slow, and hard on shared storage. Pack, send, unpack:

```
# laptop
# c create, z compress, f file name. --exclude leaves out unwanted folders
tar --exclude=.git -czf code.tgz code
scp code.tgz kelvin:~/project/
```

```
# cluster
# x extract. && means each step runs only if the last succeeded
cd ~/project && tar xzf code.tgz && rm code.tgz
```

Coming back, pack only what you need, such as summaries rather than every raw output.

4.5 Large transfers

Kelvin2 only. For large amounts of data, use a data movement node rather than a login node: `dm1.kelvin.alces.netwo` or `dm2.kelvin.alces.network`. Tools such as `rsync` can resume an interrupted transfer. MobaXterm and WinSCP offer drag and drop.

4.6 Files that arrive read-only

Copied files can keep restrictive permissions, leading to `Permission denied` when you edit them. `chmod -R u+w ~/project/code` gives you write permission on everything in that folder.

5 Software, environments and containers

You do not have administrator rights on a shared cluster, so `sudo` and system-wide installs are out. There are four main ways to get software, and it is best to try them in this order.

5.1 Step 1: Modules

Many programs and versions are installed but hidden. **Modules** make one available by changing your session's environment variables, such as `PATH`, the list of folders searched for programs.

```
# cluster
module avail                        # list everything; press space to page, q to quit
module avail 2>&1 | grep -i python  # search it; 2>&1 sends its list through the pipe
module load <name>                  # make one available
module list                         # what is loaded now
module unload <name>                # remove one
module purge                        # remove all
```

Names and versions differ between clusters and over time, so look them up rather than copying them from a guide. Put `module load` lines in your job scripts, so that each run gets the same software.

5.2 Step 2: Conda environments

An **environment** is a private folder with its own Python and packages, so that projects do not interfere with each other. **Conda** installs them, along with non-Python libraries, which matters for scientific packages such as GDAL. Work out compatible versions on a compute node, because the solve can use a lot of memory, and a login node may kill it (you will see `Killed`).

```
# cluster

# Ask for an interactive shell on a compute node: 4 cores, 16 GB, 1 hour
srun --cpus-per-task=4 --mem=16G --time=01:00:00 --pty bash

# Find the name with: module avail 2>&1 | grep -i -E "conda|forge"
module load <conda-module>

# -y answer yes; -p where to put it; -c which channel to download from
conda create -y -p ~/project/envs/sci -c conda-forge python=3.12 pandas scipy

# Back to the login node
exit
```

A conda environment can hold tens of thousands of small files, which eats into a file quota on home, so NI-HPC suggests that such installs can go on scratch. Scratch purges unused files, however, so keep a recipe: `conda env export > environment.yml` lets you rebuild it.

In a job, the most robust way to use an environment is often to call its Python by its full path:

```
# cluster, in a job script

# Clear variables that can point Python at the wrong libraries
unset PYTHONPATH PYTHONHOME
```

```

# Use the environment's own Python directly: no activate needed
~/project/envs/sci/bin/python my_script.py
```

Why the `unset`? **A Kelvin2 example.** One conda module I used set `PYTHONPATH` to its own library folder, so that Python 3.12 from my environment loaded the wrong files and crashed with `SRE module mismatch`. Clearing the variable fixed it. Environment variables quietly change how programs behave, so when something inexplicable happens, look at them with `echo "$PYTHONPATH"` or `env`.

5.3 Step 3: Compiling from source

Compiling turns human-readable source code into a program for a particular kind of machine. Compute nodes may have restricted internet access, so the usual pattern is to download on the login node and build in a job. An example in Rust, using its build tool `cargo` (Klabnik and Nichols, n.d.):

```
# cluster, login node: light work, downloads only

# Load the Rust installer
module load <rust-module>

# Keep the bulky toolchain on scratch rather than filling home
export RUSTUP_HOME=/mnt/scratch2/users/$USER/project/tmp/rustup
export CARGO_HOME=/mnt/scratch2/users/$USER/project/tmp/cargo

# Choose a version; "stable" is the current released one
rustup default stable

# Download all the libraries the project needs, without building
cd ~/project/code/myprogram && cargo fetch
```

```
#!/bin/bash
# build.sh: the build itself, as a job
#SBATCH --job-name=build
#SBATCH --time=01:00:00
#SBATCH --cpus-per-task=8            # compiling does benefit from several cores
#SBATCH --mem=16G
#SBATCH --output=logs/%x_%j.out      # %x job name, %j job ID

set -e                               # stop at the first error, rather than carrying on

module load <rust-module>
# export makes a variable visible to the programs this script starts
export RUSTUP_HOME=/mnt/scratch2/users/$USER/project/tmp/rustup
export CARGO_HOME=/mnt/scratch2/users/$USER/project/tmp/cargo
# Where the intermediate build files go: lots of small files, so scratch
export CARGO_TARGET_DIR=/mnt/scratch2/users/$USER/project/tmp/target

cd ~/project/code/myprogram
# --offline: use what was fetched; --release: optimised; -j 8: use 8 cores
cargo build --offline --release -j 8

mkdir -p ~/project/bin
cp "$CARGO_TARGET_DIR/release/myprogram" ~/project/bin/
echo BUILD OK
```

Check the result before you trust it. Run a case with a known answer, ideally one you have also run elsewhere, and compare the numbers.

5.4 Step 4: Containers

5.4.1 What containerisation is

A **container** packages a program with the exact libraries it depends on and runs it isolated from the rest of the system. Everything lives in an **image**, a read-only template built from a recipe. Run an image and you get a container.

Think of a shipping container: what is inside does not depend on the ship. "It works on my machine" becomes "it works wherever this image runs". This helps with reproducibility, and with software that is hard to install.

A container is not a virtual machine. A virtual machine emulates a whole computer, with its own operating system kernel. A container shares the host's kernel, so it starts quickly and with little overhead (Merkel, 2014).

Dockerisation means packaging your application as a Docker image, through a recipe called a `Dockerfile`:

```
# FROM: start from a ready-made image
FROM python:3.12-slim
# COPY: put our code inside the image
COPY heat_loss.py /app/heat_loss.py
# ENTRYPOINT: what to run when the container starts
ENTRYPOINT ["python3", "/app/heat_loss.py"]
```

```
# laptop, with Docker Desktop running

# Build an image from the Dockerfile in this folder (.), and name it (-t)
docker build -t heatloss .

# Run it. --rm tidies up afterwards. -v shows a laptop folder inside the container as /data
docker run --rm -v "$(pwd):/data" heatloss /data/case_000.json
```

5.4.2 Why not Docker on a cluster?

Docker normally relies on a background service with root (administrator) privileges. On a shared system, that is a security problem, so clusters rarely provide it (Kurtzer et al., 2017).

Apptainer (formerly Singularity) was designed for this setting. It runs as an ordinary user, and a container is a single file that you can copy around. It can also convert Docker images:

```
# cluster; check the module name with: module avail 2>&1 | grep -i apptainer
module load <apptainer-module>

# Convert an image from Docker Hub into a single file, heatloss.sif
apptainer pull heatloss.sif docker://<dockerhub-user>/heatloss

# Run it. --bind makes a cluster folder visible inside the container
apptainer run --bind "$PWD:/data" heatloss.sif /data/case_000.json
```

Which to use? If a module exists, use it, as central builds are often tuned for the machine. If a Python stack is all you need, conda is simple. If you compiled something once and it works, that is fine too. Use a container when a program is hard to install, or when others must reproduce your exact software. See the Apptainer documentation[1] for more.

[1] `https://apptainer.org/docs/user/latest/`

6 Running jobs with Slurm

6.1 Two ways to run

	`srun ... --pty bash`	`sbatch script.sh`
Style	Interactive: you get a shell on a compute node	Hand it in and walk away
Good for	Short tests, installs	All real work
If you disconnect	Work stops	It carries on

6.2 A first job

Save this as `first.sh`:

```
#!/bin/bash
# The first line (the "shebang") says which program should run this file.

# Lines starting #SBATCH are instructions to Slurm. They must come before any command.
#SBATCH --job-name=first             # a label for the queue
#SBATCH --time=00:05:00              # limit: hours:minutes:seconds. The job is stopped at this point
#SBATCH --cpus-per-task=2            # cores for this job
#SBATCH --mem=2G                     # memory for this job
#SBATCH --output=first_%j.out        # where printed output goes; %j becomes the job ID

hostname                             # which node am I on?
nproc                                # how many cores did I get?
date                                 # when did it run?
```

```
# cluster
sbatch first.sh               # hand it in. Prints: Submitted batch job <jobid>
squeue -u $USER               # my jobs. PD means pending, R means running
cat first_<jobid>.out         # read what it printed
```

When your job vanishes from `squeue`, it has ended, either successfully or not, so always read the log. By default a job starts in the folder from which you submitted it.

6.3 The settings you need

Setting	Meaning	Choosing it
`--job-name`	A label	Short and useful
`--time`	Hard limit, also `days-hh:mm:ss`	Run a test, and add a margin. An accurate limit helps a job start sooner
`--cpus-per-task`	Cores for a threaded program	Only what it can actually use
`--ntasks`	Separate processes, as in MPI	1, unless you use MPI
`--nodes`	Number of nodes	Usually leave it, unless using MPI
`--mem` or `--mem-per-cpu`	Memory	Check real use with `sacct`, then trim it
`--partition` or `-p`	Which queue	See below

Setting	Meaning	Choosing it
`--output`, `--error`	Files for printed output and errors	Use `%x` job name, `%j` job ID, `%A` array ID, `%a` task number. The folder must already exist, or the log is lost
`--array`	Many tasks; see the next chapter	
`--gres=gpu:<type>:1`	A GPU	Only on GPU partitions, and only if your code uses it
`--mail-type`, `--mail-user`	Email on start and end	Handy for long jobs; avoid it for arrays

The definitive list is in the `sbatch` manual[1].

Partitions, **Kelvin2 only**, from Running Jobs[2]; check there for changes:

Partition	Time limit	For
`k2-hipri` (default)	3 hours	Short jobs
`k2-medpri`	24 hours	Longer jobs
`k2-lowpri`	Not listed	Long jobs
`k2-himem`	3 days	Very large memory
`k2-gpu-*`	3 days	GPUs
`k2-sandbox`	See `sinfo`	Testing, per the Green HPC page

6.4 Watching and managing

```
# cluster
squeue -u $USER                 # my queued and running jobs
squeue -u $USER -h | wc -l      # -h removes the header, so this counts my jobs
scancel <jobid>                 # cancel a job
scancel -u $USER                # cancel all my jobs. Careful
sinfo -s                        # a summary of partitions
sacct -j <jobid> --format=JobID,Elapsed,MaxRSS,State    # after it ends: time, peak memory, outcome
```

Job states you will see include `COMPLETED`, `FAILED`, `TIMEOUT` (it hit `--time`), and `OUT_OF_MEMORY`.

6.5 Start small

Test on a tiny case before running at scale. A good first job proves that it runs on a compute node (`hostname`), has the cores you asked for (`nproc`), finds your software, and can see your files. NI-HPC's Green HPC page[3] makes the same point on carbon grounds: failed jobs waste energy too.

[1] `https://slurm.schedmd.com/sbatch.html`
[2] `https://ni-hpc.github.io/nihpc-documentation/Running%20Jobs/`
[3] `https://ni-hpc.github.io/nihpc-documentation/Green%20HPC/`

7 Seven exercises to build confidence

The fastest way to feel at home on a cluster is to run small jobs that cannot go wrong for long. Each exercise here takes a minute or two and adds one new idea. You compare your output with the answer given, so you know whether it worked.

They need only Python's standard library, so there is nothing to install. Every job has a short time limit and asks for little memory, so it should start quickly and costs almost nothing.

7.1 Get the files

Download the starter kit[1] to your computer, or fetch it directly on the cluster:

```
# cluster
cd ~
# -O keeps the file name from the address
curl -LO https://ahmadammar.github.io/teaching/hpc/examples/hpc-starter-kit.zip
unzip hpc-starter-kit.zip
cd hpc-starter-kit
# Slurm will not create a log folder for you
mkdir logs
```

Your cluster may need extra settings, such as a partition or account name. If a job is refused, the error message and your cluster's documentation will say which. If `python3` is not found, run `module avail python` and add the right `module load` line to the script.

7.2 Exercise 1: Hello from a compute node

New idea: a job runs on a different computer from the one you typed on.

01_hello.sh

```
#!/bin/bash
#SBATCH --job-name=hello
#SBATCH --time=00:02:00
#SBATCH --cpus-per-task=1
#SBATCH --mem=500M
#SBATCH --output=logs/%x_%j.out

# The simplest useful job: where am I, when is it, and can I do a sum?
echo "Hello from $(hostname)"
echo "The time is $(date)"
echo "Job $SLURM_JOB_ID, submitted from $SLURM_SUBMIT_DIR"
echo "1 + 1 = $(( 1 + 1 ))"
```

```
# cluster
hostname
sbatch 01_hello.sh
# Watch it until it disappears from the queue
squeue -u $USER
# The log is named after the job and its number
cat logs/hello_*.out
```

[1] https://ahmadammar.github.io/teaching/hpc/examples/hpc-starter-kit.zip

Check: the log ends with `1 + 1 = 2`, and the name in `Hello from` differs from what `hostname` printed on the login node. That difference is the whole idea of a batch system.

7.3 Exercise 2: Run a small program

New idea: a job can run your own code, with arguments.

02_add.py

```
"""Add two numbers given on the command line."""
import sys

a, b = float(sys.argv[1]), float(sys.argv[2])
print(f"{a} + {b} = {a + b}")
```

```
# cluster
sbatch 02_add.sh
cat logs/add_*.out
```

Check: you see your Python version, then `1.0 + 1.0 = 2.0` and `21.0 + 21.0 = 42.0`.

Try: break it on purpose. Change the script to run `02_add.py 1` with one number, submit it and read the error in the log. Then run `sacct -X -j <job id>`. You have now seen a failed job, and how to find out why.

7.4 Exercise 3: Ask for more cores

New idea: more cores help only if the program uses them, and the gain is rarely perfect.

03_primes.py

```
"""Count primes below a limit, on as many cores as Slurm gave this job. Standard library only."""
import multiprocessing as mp
import os
import sys
import time


def is_prime(n):
    if n < 2:
        return False
    return all(n % d for d in range(2, int(n ** 0.5) + 1))


if __name__ == "__main__":
    limit = int(sys.argv[1]) if len(sys.argv) > 1 else 1_000_000
    cores = int(os.environ.get("SLURM_CPUS_PER_TASK", 1))
    start = time.time()
    with mp.Pool(cores) as pool:
        count = sum(pool.map(is_prime, range(limit), chunksize=10_000))
    print(f"{count} primes below {limit}, on {cores} core(s), in {time.time() - start:.1f} s")
```

```
# cluster
sbatch 03_primes.sh
# Change --cpus-per-task to 1 in the script, then submit it again
sbatch 03_primes.sh
grep primes logs/primes_*.out
```

Check: every run reports `78498 primes below 1000000`, the known value. Only the time changes. On a test laptop, 1 core took 16.9 s and 4 cores took 7.7 s: 2.2 times faster, not 4. Your times will differ, because splitting and collecting work takes time too.

7.5 Exercise 4: Analyse real data

New idea: data travels with the job, and a relative path works because the job starts in the folder you submitted from.

The data is the monthly record from Armagh Observatory in Northern Ireland, published by the Met Office. It is real, so it has gaps: early years lack temperatures, and only complete years count.

```
# cluster
head -8 data/armaghdata.txt
sbatch 04_armagh.sh
cat logs/armagh_*.out
```

The first command shows the top of the data file, which is how you learn its layout:

What you should see from head

```
Armagh
Location: 287800E 345800N (Irish Grid), Lat 54.352 Lon -6.649, 62 metres amsl
Estimated data is marked with a * after the value.
Missing data (more than 2 days missing in month) is marked by  ---.
Sunshine data taken from an automatic Kipp & Zonen sensor marked with a #, otherwise sunshine data taken from a
     Campbell Stokes recorder.
   yyyy  mm   tmax    tmin      af    rain     sun
              degC    degC    days      mm   hours
   1853   1    ---     ---     ---    57.3     ---
```

Check, for the file in the kit:

```
2085 monthly records, 159 complete years, 1867 to 2025
Warmest: 2023, mean 10.95 C
Coldest: 1892, mean 7.89 C
Wettest: 1872, 1249 mm
```

7.6 Exercise 5: Your first job array

New idea: one submission can start many tasks, and each uses its number to choose its own work. This is how most research sweeps run.

05_decade.py

```
"""One array task handles one decade of the Armagh record, and writes one small result file."""
import os
from armagh import read_months, yearly

FIRST = 1870
task = int(os.environ.get("SLURM_ARRAY_TASK_ID", 0))
decade = FIRST + 10 * task
years = yearly(read_months("data/armaghdata.txt"))
values = [v["tmean"] for y, v in years.items() if decade <= y < decade + 10]
os.makedirs("results", exist_ok=True)
with open(f"results/decade_{decade}.csv", "w") as f:
    f.write(f"{decade},{len(values)},{sum(values) / len(values):.2f}\n")
print(f"task {task}: {decade}s, {len(values)} years")
```

```
# cluster
sbatch 05_decades.sh
# Each task appears with its number, such as 12345_3
squeue -u $USER
# After all finish, count the results: expect 15
ls results/decade_*.csv | wc -l
# Join them, with the coolest decade first and the warmest last
cat results/decade_*.csv | sort -t, -k3 -n
```

Each line reads decade, years used, mean temperature. **Check:** the coolest is `1880,10,8.65` and the warmest is `2000,10,10.08`.

What you should see

```
$ cat results/decade_*.csv | sort -t, -k3 -n
1880,10,8.65
1890,10,8.87
1900,10,8.91
1910,10,8.94
1920,10,9.08
1960,10,9.11
1970,10,9.12
1980,10,9.14
1930,10,9.27
1950,10,9.35
1870,10,9.52
1940,10,9.58
1990,10,9.71
2010,10,9.87
2000,10,10.08
```

These results come from running the same Python scripts on a laptop, with the task number set by hand. On the cluster, the same numbers appear, in the files or logs named above.

The pattern to remember is that the tasks never talk to each other. Each writes its own file, and a simple command joins them at the end.

7.7 Exercise 6: A geospatial calculation

New idea: an input can come from the command line, so one script can answer many questions.

The data is about 240 cities from Natural Earth, a public domain map dataset. The script uses the haversine formula for distance over a sphere.

```
# cluster
sbatch 06_cities.sh
cat logs/cities_*.out
```

Check:

```
243 cities. Nearest to London:
  The Hague: 309 km
  Brussels: 319 km
  Paris: 343 km
Farthest: Wellington, 18815 km
```

Try: pick another city from `data/cities_natural_earth.csv`. Do the distances look right on a map?

7.8 Exercise 7: Satellite tiles in parallel

New idea: large images are usually cut into tiles and processed one tile per task.

The Normalised Difference Vegetation Index measures greenness from red and near-infrared light:

```
NDVI = (NIR - red) / (NIR + red)
```

These tiles are **synthetic**: a script invents them so that the exercise runs anywhere. They are built to run from built-up land (tile 0) to parkland (tile 7).

```
# cluster
# Light setup work like this is fine on a login node
python3 07_make_tiles.py
sbatch 07_tiles.sh
# When it has finished
cat results/ndvi_*.csv
```

Check: mean NDVI rises steadily from `0.045` in tile 0 to `0.787` in tile 7. Run it again and the numbers stay the same, because the script fixes its random seed.

What you should see

```
$ python3 07_make_tiles.py
wrote 8 tiles to tiles/

$ cat results/ndvi_*.csv
0,10000,0.045
1,10000,0.181
2,10000,0.304
3,10000,0.416
4,10000,0.522
5,10000,0.616
6,10000,0.705
7,10000,0.787
```

Each line reads tile, pixels, mean NDVI. These results come from running the same Python scripts on a laptop, with the task number set by hand. On the cluster, the same numbers appear, in the files or logs named above.

7.9 From exercises to your own project

You have now used every piece of a real workflow. To move your own work across:

1. Run your program once on your own computer, or in an interactive session on the cluster, on the smallest input you can.
2. Submit that as one job. Compare the answer with the known one.
3. Use `sacct` to see the time and memory it really used, then set your limits a little above that.
4. Run a few tasks of the array, not all of them: `sbatch --array=0-3`.
5. Check that the outputs make sense and can be restarted. Then run the rest.

The next chapter does this at full size, with a parameter sweep.

8 Job arrays: a parameter sweep

Say you want to run one program on hundreds of inputs. Submitting hundreds of separate jobs is clumsy. A **job array** submits them as one, with each **task** told its number (SchedMD, n.d.).

This chapter builds a working sweep. Download the three files: `heat_loss.py`, `make_cases.py` and `run_sweep.sh`. The "simulation" estimates a building's yearly heating need, but you can swap in your own program.

8.1 The science in one paragraph

Heat flows out through a wall in proportion to its area, how leaky it is (its **U-value**, in W per m^2 per kelvin) and the temperature difference. Air leaking out takes heat too: warming 1 m^3 of air by 1 K takes about 0.33 Wh. **Degree days** measure how cold a year is, summing how far each day falls below a base temperature. So heat in kWh per year is:

$$Q = (U \times A + 0.33 \times n \times V) \times DD \times 24 / 1000$$

where A is wall area, n is air changes per hour, V is volume and DD is degree days.

8.2 Step 1: the program

heat_loss.py

```
#!/usr/bin/env python3
"""Toy simulation for the HPC tutorial: the yearly heating need of one simple building.

    python heat_loss.py case_007.json

It reads one case, does a small calculation, and writes the answer next to the input.
The physics is deliberately basic (see "heat loss" in the tutorial). The numbers are
illustrative: the point is to have something small that you can run thousands of times.
"""
import json   # reads and writes JSON, a plain-text format of names and values
import sys    # gives access to the words typed after the script name (sys.argv)
from pathlib import Path   # handles file paths in a tidy way on any operating system


def yearly_heat(u_wall, wall_area, air_changes, volume, degree_days):
    """Return heat need in kWh per year.

    u_wall       how leaky the wall is, in W per square metre per kelvin
    wall_area    wall area, in square metres
    air_changes  how many times per hour the indoor air is replaced
    volume       indoor volume, in cubic metres
    degree_days  how cold the year is, in kelvin-days
    """
    fabric = u_wall * wall_area               # heat lost through the walls, W per kelvin
    air = 0.33 * air_changes * volume         # heat lost with the air; 0.33 Wh warms 1 m3 of air by 1 K
    return (fabric + air) * degree_days * 24 / 1000   # W/K x K-days x 24 h/day = Wh; /1000 gives kWh


def main():
    path = Path(sys.argv[1])                  # sys.argv[1] is the first word after the script name
```

```
    case = json.loads(path.read_text())        # read the file, and turn its text into a Python dictionary
    answer = yearly_heat(**case)               # ** passes the dictionary's entries as named arguments
    # with_suffix swaps .json for .out, so case_007.json gives case_007.out
    path.with_suffix(".out").write_text(f"{answer:.1f}\n")   # :.1f means one decimal place


if __name__ == "__main__":                     # true only when run directly, not when imported
    main()
```

8.3 Step 2: make the inputs

make_cases.py

```
#!/usr/bin/env python3
"""Write one input file per combination of settings: a small parameter sweep.

    python make_cases.py cases
"""
import itertools   # itertools.product gives every combination of several lists
import json
import sys
from pathlib import Path

U_WALL = [2.1, 1.5, 0.45]        # three wall types: W per square metre per kelvin
AIR_CHANGES = [1.0, 0.7, 0.5]    # three levels of draughtiness, per hour
DEGREE_DAYS = [1800, 2000, 2200] # three climates, kelvin-days

folder = Path(sys.argv[1])       # the folder name typed after the script name
folder.mkdir(parents=True, exist_ok=True)   # make it, with no error if it exists already

# enumerate numbers the combinations 0, 1, 2 ... so that each file gets a unique name
for number, (u, ach, hdd) in enumerate(itertools.product(U_WALL, AIR_CHANGES, DEGREE_DAYS)):
    case = {"u_wall": u, "wall_area": 90, "air_changes": ach, "volume": 200, "degree_days": hdd}
    # :03d pads the number to three digits, so that files sort in order: 000, 001 ... 026
    (folder / f"case_{number:03d}.json").write_text(json.dumps(case))

print(f"wrote {number + 1} cases to {folder}")
```

Three settings, each with three values, give 3 × 3 × 3 = 27 combinations.

```
# cluster, in a project folder holding the three files

# Find your Python first. Look for a module with: module avail 2>&1 | grep -i python
module load <python-module>
python3 make_cases.py cases
ls cases | head                # case_000.json, case_001.json ...
python3 heat_loss.py cases/case_000.json && cat cases/case_000.out    # 11016.0
```

What you should see

```
$ python3 make_cases.py cases
wrote 27 cases to cases

$ ls cases | head
case_000.json
case_001.json
case_002.json
case_003.json
case_004.json
case_005.json
case_006.json
case_007.json
```

```
case_008.json
case_009.json

$ python3 heat_loss.py cases/case_000.json && cat cases/case_000.out
11016.0
```

The last line is your test. For the first case, U = 2.1, A = 90, n = 1, V = 200 and DD = 1800, so by hand: (189 + 66) × 1800 × 24 / 1000 = 11016. Then delete that output, with `rm cases/case_000.out`, so that the job below runs every case.

8.4 Step 3: the job script

Running one input per task works, but very short tasks are wasteful, because starting each one has a cost. So **pack** several runs into each task.

run_sweep.sh

```
#!/bin/bash
# Runs a block of cases on one core, as one task of a Slurm job array.
#   sbatch --array=0-5 run_sweep.sh cases 5
# means: 6 tasks (numbered 0 to 5), with 5 cases per task, from the folder "cases".

# Lines starting #SBATCH are read by Slurm. To bash, they are just comments.
#SBATCH --job-name=sweep            # the label shown in the queue
#SBATCH --time=00:10:00             # hard limit, hours:minutes:seconds; the job is stopped when it is reached
#SBATCH --cpus-per-task=1           # one core: this program cannot use more
#SBATCH --mem=1G                    # memory for each task
#SBATCH --output=logs/%x_%A_%a.out  # %x job name, %A array job ID, %a task number; make logs/ first

IN="$1"                             # $1 is the first word after the script name
PER="${2:-5}"                       # $2 is the second word; :-5 means "use 5 if it is missing"

# Fill a bash array named CASES with the sorted file names.
# < <( ... ) feeds the output of a command into mapfile; -t removes the line endings.
mapfile -t CASES < <(ls "$IN"/case_*.json | sort)

# Slurm sets SLURM_ARRAY_TASK_ID for each task. $(( )) does whole-number arithmetic.
# Task 0 starts at case 0, task 1 at case 5, task 2 at case 10, and so on.
start=$(( SLURM_ARRAY_TASK_ID * PER ))

# ${CASES[@]:start:PER} means: PER items of the array, beginning at position start.
for f in "${CASES[@]:start:PER}"; do
  # ${f%.json} removes the ending .json. -f tests whether a file exists.
  # && continue skips to the next case if this one is done, so resubmitting is safe.
  [ -f "${f%.json}.out" ] && continue
  # || runs the echo only if python failed, leaving a line you can search for.
  python3 heat_loss.py "$f" || echo "FAILED $f"
done

echo "task $SLURM_ARRAY_TASK_ID finished its block, starting at case $start on $(hostname)"
```

Add your `module load` line just before the `mapfile` line. Slurm usually passes your session's environment on to the job, but stating it in the script is what makes the job repeatable.

8.5 Step 4: how many tasks?

With N cases and P per task, divide and round up, so that the last few cases are covered. Tasks are numbered from 0. For 27 cases at 5 per task, that is 6 tasks, numbered 0 to 5.

```
# cluster

# Line endings, if you edited on Windows
sed -i 's/\r$//' run_sweep.sh

# The log folder must exist before submission
mkdir -p logs

# Count the cases, and work out the last task number
N=$(ls cases/case_*.json | wc -l)
PER=5
# In whole-number arithmetic, (N + P - 1) / P rounds up
LAST=$(( (N + PER - 1) / PER - 1 ))
echo "$N cases, tasks 0 to $LAST"

sbatch --array=0-$LAST run_sweep.sh cases $PER
```

What you should see from the echo line

```
27 cases, tasks 0 to 5
```

The `sbatch` line then prints the number of your job, which differs each time.

A useful extra: `--array=0-99%10` runs at most 10 tasks at once, which is polite on a busy system.

8.6 Step 5: did it all work?

```
# cluster
squeue -u $USER                      # wait until it is empty
ls cases/*.out | wc -l               # expect 27
grep -l FAILED logs/*.out | wc -l    # expect 0
tail -2 logs/sweep_*_0.out           # read one log
```

If some failed, resubmit the same command. Because each task skips cases that already have an output, only the missing ones run. Being safe to re-run like this makes large runs much less stressful.

8.7 Step 6: collect the results

```
# cluster
# A loop that prints each case's name and answer into a CSV file.
# ${f%.out} strips the ending, and $(cat "$f") inserts the file's contents
for f in cases/*.out; do echo "$(basename "${f%.out}"),$(cat "$f")"; done > results.csv
head -3 results.csv
```

What you should see from head

```
case_000,11016.0
case_001,12240.0
case_002,13464.0
```

```
# laptop
scp kelvin:~/project/results.csv "D:\My Work"
```

8.8 Choosing a packing

	Few runs per task (many tasks)	Many runs per task (few tasks)
Finishes	Sooner, if cores are free	Later: each task takes longer
Overhead	Higher: more starts, more load on the scheduler	Lower
After a failure	Less lost	More to redo, unless it is restart-safe

There is no single right answer. Test with your own program, with care: submit each setting separately, repeat it, and keep the copying of files out of your timing. If you are planning a very large run, ask your support team what they prefer.

9 Measuring and good practice

9.1 Words for time

Term	Meaning
Wall time	Real time by the clock on the wall
Core time	Sum of the time on each core. 10 tasks of 1 minute each: 1 minute of wall time, 10 of core time
Queue wait	From submitting to starting
Speed-up	Old wall time / new wall time
Efficiency	Speed-up / number of cores used

9.2 Slurm's accounts

Slurm records everything, and `sacct` reads it back.

```
# cluster

# -X: whole jobs only. -P: separate columns with |, ready for a spreadsheet or pandas.
# -S: from this date. -o: which columns. > saves to a file
sacct -X -P -S 2026-01-01 -u $USER -o JobID,JobName,Submit,Start,End,Elapsed,State > jobs.txt
```

Times are in whole seconds, which is too coarse for tasks of a few seconds. Time those from inside the job with `date +%s.%N`.

9.3 A measured example

The same prime-counting program from the exercises, up to 5 million, was run three times at each core count on Kelvin2 and on one laptop (Intel Core i5-10210U, 4 physical cores, doing other work at the time). Each cell is the middle of three runs. All 27 runs found the same answer, 348,513.

Cores	Kelvin2	Speed-up	Efficiency	Laptop	Laptop range
1	46.0 s	1.0x	100%	209 s	125 to 255 s
2	23.2 s	2.0x	99%	125 s	82 to 156 s
4	12.0 s	3.8x	96%	69 s	66 to 105 s
8	6.3 s	7.3x	91%	61 s	55 to 66 s
16	3.7 s	12.4x	78%	n/a	n/a

Three things are worth noticing:

- **Efficiency falls as cores rise**, because splitting and collecting work takes time. Sixteen cores gave 12.4 times the speed, not 16.
- **The laptop barely gains beyond 4 cores.** It has 4 physical cores, and its 8 are logical threads sharing them. Its times also vary widely, because it was busy, while Kelvin2's stayed within a fraction of a second. So don't read a single speed-up figure from this comparison.

- **Many jobs can run at once.** The 15 Kelvin2 jobs were submitted together in 2026, waited 10 seconds, and finished within 57 seconds. One after another they would take about 4.6 minutes.

This program is well suited to splitting up. Yours may not be, so measure it.

To repeat the test:

```
# cluster, in the starter kit folder
for r in 1 2 3; do for c in 1 2 4 8 16; do
  sbatch --job-name=t${c}_${r} --cpus-per-task=$c --mem=4G --time=00:30:00 \
    --output=logs/timing_${c}_${r}.out --wrap="python3 03_primes.py 5000000"
done; done
```

9.4 Benchmarking fairly

Comparing a cluster with a laptop is easy to do badly.

- Compare like with like: same program, version, input and output settings.
- Report what you measured. "First task start to last task end" and "submission to finish" are different numbers.
- Separate setup from running. If your script copies files between submissions, you are timing the copying. I made this mistake in my own first test.
- Repeat. Shared systems vary with load. Report a range, not your best run.
- Say what you projected, and what you measured.
- Check that the answers match, not just the speed.

9.5 Good habits

These follow the spirit of Wilson et al. (2014).

- Keep code in version control, such as git. Keep raw data unchanged, with scripts that make everything else from it.
- Never overwrite results. Use dated output folders: `outputs/$(date +%Y-%m-%d_%H%M)`.
- Record versions: `module list`, `conda env export`, `git rev-parse HEAD`.
- Keep job scripts, as they are the record of what you did.
- Make jobs restartable.
- Read your inputs back: check a run's settings, such as the simulated period, before you describe it.
- Copy results off scratch.

9.6 Being a good neighbour

- No heavy work on login nodes.
- Ask for what you need, not the maximum. Over-asking makes your own job queue for longer, and blocks others.
- Avoid millions of tiny files. Pack them, or write summaries.

- Run only what you need. Smarter sampling can replace brute force, as NI-HPC's Green HPC page[1] notes.
- Read your cluster's acceptable use policy.
- Acknowledge the service in your papers. **Kelvin2:** "We are grateful for use of the computing resources from the Northern Ireland High Performance Computing (NI-HPC) service funded by EPSRC (EP/T022175)."
- When stuck, write to the support team with your job ID, the exact command and the error message.

[1] `https://ni-hpc.github.io/nihpc-documentation/Green%20HPC/`

10 Other clusters, the cloud and Linux basics

Nearly everything in this guide carries over. A new system mostly changes the names, the storage rules and who pays. This chapter is a map to help you adapt, and each system's own documentation has the final word.

10.1 A checklist for any new system

1. Who can use it, and how do I apply?
2. How do I log in: which address, port, keys and multi-factor steps?
3. Which storage is backed up, which is fast, and how much do I have?
4. Which scheduler does it use, and which partitions or queues?
5. How is software provided: modules, containers, or my own installs?
6. What are the limits on time, memory and number of jobs?
7. How should I acknowledge it in publications?

Then run Exercise 1 of this guide. If that works, the rest will follow.

10.2 Other schedulers

Slurm is common but not universal. The ideas match, while the words differ.

Task	Slurm	PBS / Torque	Grid Engine	LSF
Submit	`sbatch`	`qsub`	`qsub`	`bsub`
See the queue	`squeue`	`qstat`	`qstat`	`bjobs`
Cancel	`scancel`	`qdel`	`qdel`	`bkill`
Script directive	`#SBATCH`	`#PBS`	`#$`	`#BSUB`
Array task number	`$SLURM_ARRAY_TASK_ID`	`$PBS_ARRAYID` or `$PBS_ARRAY_INDEX`	`$SGE_TASK_ID`	`$LSB_JOBINDEX`

10.3 Cluster or cloud?

	Shared cluster	Cloud
Cost	Usually free or allocated to researchers	Paid by use, so a forgotten machine keeps charging
Waiting	A queue	Usually starts at once
Control	Fixed system; no administrator rights	You control the machine, and so must secure and maintain it
Best for	Large batches, shared storage	Special hardware, short bursts, web services

A cloud virtual machine feels familiar: you create it, log in with SSH using the same kind of key, and run commands on Linux. The usual pattern is:

1. Set a budget alert before you start anything.
2. Create the smallest machine that does the job, and allow SSH only from your own address where you can.
3. Keep data in object storage, such as S3-style buckets, as a machine's own disk may be deleted with it.
4. Run long work inside `tmux`, described below, so that it survives a dropped connection.
5. Stop or delete the machine when you finish. Check that its disks and addresses are gone too, as they can keep charging.

For many tasks, services such as AWS Batch, Azure Batch and Google Cloud Batch offer something like a job array, and some providers can build a Slurm cluster for you. Containers from the software chapter move well between all of these.

10.4 Linux basics for looking after your own work

10.4.1 Permissions

Every file has an owner and a group, with read (`r`), write (`w`) and execute (`x`) rights for each.

```
# any Linux system
# Shows, for example, -rwxr-x--- : owner all, group read and run, others nothing
ls -l run.sh
# Let yourself run a script
chmod u+x run.sh
# Keep a private SSH key readable only by you. SSH refuses keys left more open
chmod 600 ~/.ssh/my-key
```

10.4.2 Space

```
# any Linux system
# How full are the disks?
df -h
# What is using the space here, biggest last?
du -sh * | sort -h
```

Many clusters have their own command for your quota. Check its documentation.

10.4.3 Processes

```
# any Linux system
# A live view of what is running. Press q to leave
top
# Just your own programs
ps -u $USER
# Stop a process politely, using its number from ps
kill <process id>
```

On a cluster, stop jobs with `scancel`, not `kill`.

10.4.4 Sessions that survive a disconnection

`tmux` keeps a terminal running on the remote machine after you leave.

```
# any Linux system
tmux new -s work
# Do your work, then press Ctrl-b, then d, to leave it running
# Come back later
tmux attach -t work
```

On a cluster, use it for light work on login nodes. Heavy work still belongs in jobs.

10.4.5 Scheduled tasks

`cron` runs commands at set times on a machine you manage. Edit your list with `crontab -e`. This line runs a backup at 02:00 every day:

```
0 2 * * * /home/me/backup.sh >> /home/me/backup.log 2>&1
```

Check whether your cluster allows it before relying on it.

10.4.6 Installing system software

On your own machine or cloud server, you have administrator rights through `sudo`:

```
# Debian or Ubuntu
sudo apt update && sudo apt install tmux
# Red Hat, Rocky or Fedora
sudo dnf install tmux
```

On a shared cluster you will not have `sudo`. Use modules, conda or containers.

10.4.7 Keeping your own server safe

If you run a cloud machine, its security is your job. Log in with keys and not passwords, apply updates regularly, open only the network ports you need, keep secrets out of git, and keep backups somewhere else. Test that you can restore them.

11 Troubleshooting and cheat sheet

11.1 Errors and what they mean

Symptom	Likely cause	Fix
`Permission denied (publickey)`	Wrong key, username or port, or the key is not installed yet	Check `.ssh/config`, and that the cluster has your key
Host identification warning	A login node you have not met, or a real problem	Read your cluster's guidance before accepting
`scp`: "No such file" with a stray `"`	Trailing backslash in a Windows path	Remove the backslash
`scp`: wrong port	`-p` instead of `-P`	`scp` uses capital `-P`
`Killed` on the login node	Too much memory on a shared node	Use `srun` or `sbatch`
`CondaError: Run 'conda init'`	Shell not set up for conda	`conda init`, then log in again; or call the environment's Python directly
`EnvironmentLocationNotFound`	The environment was never created, often as an earlier step failed	Scroll up and fix the first error
`SRE module mismatch` or strange Python errors	`PYTHONPATH` points at another Python's libraries	`unset PYTHONPATH PYTHONHOME`
`No module named ...` in a job	The job is using a different Python	Call your Python by its full path
`rustup` says no default toolchain	Only the installer is loaded	`rustup default stable`
Downloads fail in a job	Compute nodes may have restricted internet	Download on the login node first, and build offline
`Permission denied` on your own files	Read-only after copying	`chmod -R u+w folder`
`$'\r': command not found`	Windows line endings	`sed -i 's/\r$//' script.sh`
No log file appears	The `--output` folder does not exist	`mkdir -p logs` before you submit
State `TIMEOUT`	Hit the time limit	More `--time`, or a longer partition
State `OUT_OF_MEMORY`	Hit the memory limit	More `--mem`
Stuck on `PD` for ages	Busy, or you asked for too much	`squeue -u $USER` shows the reason; ask for less
Files vanished from scratch	Purged after no access	Keep copies elsewhere
Pasted lines ran in an odd order	Multi-line paste in a terminal	Paste one line at a time and check

The general method: read the **first** error rather than the last, as later errors are often consequences. Then search for its exact text.

11.2 Cheat sheet

```
# cluster: daily use
squeue -u $USER                                 # what is running
sbatch job.sh                                   # submit
sbatch --array=0-9 job.sh args                  # submit an array
scancel <jobid>                                 # cancel
tail -20 "$(ls -t logs/*.out | head -1)"        # newest log
sacct -X -P -S 2026-01-01 -u $USER -o JobID,JobName,Elapsed,State   # history
srun --cpus-per-task=1 --mem=4G --time=00:15:00 --pty bash          # interactive
module avail; module load <name>; module list  # software
quota -s; du -sh ~/project                      # storage
tar czf out.tgz folder; tar xzf out.tgz         # pack and unpack
```

```
# laptop
ssh kelvin                                      # log in
```

```
scp file kelvin:~/project/                    # upload
scp kelvin:~/project/file "D:\My Work"        # download
```

11.3 References

Amdahl, G.M. (1967) 'Validity of the single processor approach to achieving large scale computing capabilities', *AFIPS Spring Joint Computer Conference*, pp. 483-485. https://doi.org/10.1145/1465482.1465560

Klabnik, S. and Nichols, C. (n.d.) *The Rust Programming Language*. https://doc.rust-lang.org/book/

Kurtzer, G.M., Sochat, V. and Bauer, M.W. (2017) 'Singularity: scientific containers for mobility of compute', *PLoS ONE*, 12(5), e0177459. https://doi.org/10.1371/journal.pone.0177459

McNabb, R. (n.d.) *Introduction to high performance computing*. https://iamdonovan.github.io/teaching/hpc-intro/index.html

Merkel, D. (2014) 'Docker: lightweight Linux containers for consistent development and deployment', *Linux Journal*, 2014(239).

NI-HPC (n.d.) *NI-HPC documentation*. https://ni-hpc.github.io/nihpc-documentation

OpenSSH (n.d.) *OpenSSH manual pages*. https://www.openssh.com/manual.html

Raicu, I., Foster, I.T. and Zhao, Y. (2008) 'Many-task computing for grids and supercomputers', *Workshop on Many-Task Computing on Grids and Supercomputers*, pp. 1-11. https://doi.org/10.1109/MTAGS.2008.4777912

SchedMD (n.d.) *Slurm job array support*. https://slurm.schedmd.com/job_array.html

Shotts, W. (2019) *The Linux Command Line*. 5th internet edn. https://linuxcommand.org/tlcl.php

Wilson, G., Aruliah, D.A., Brown, C.T., et al. (2014) 'Best practices for scientific computing', *PLoS Biology*, 12(1), e1001745. https://doi.org/10.1371/journal.pbio.1001745

Yoo, A.B., Jette, M.A. and Grondona, M. (2003) 'SLURM: Simple Linux Utility for Resource Management', in *Job Scheduling Strategies for Parallel Processing*. Springer, pp. 44-60. https://doi.org/10.1007/10968987_3